Scenix Semiconductor, Inc Virtual Peripheral Guidelines: 0.997

Chris Fogelklou
& the
Applications Team
Scenix, Inc.
December 14, 1999

Index of Proposed Guidelines:

Page #.- Subject.

Index of Proposed Guidelines:

Page #.- Subject.

INTRODUCTION

This document describes the formats, conventions, and coding guidelines to make integrating various Virtual Peripherals with one another as "cut and paste" for a beginning Scenix programmer as is possible.

A Virtual Peripheral is a software peripheral that interacts with the Application Software the same way a hardware peripheral would. Implementing peripherals in software speeds development time and decreases a product's time to market.

The concept of Virtual Peripherals has been around for decades, but the hardware to make real the concept did not exist in a practical form until Scenix Semiconductor released the Scenix SX processor in 1997. The SX processor is an inexpensive RISC 8-bit microcontroller, running at up to 100MIPS of performance. Although speed is nice, the true Virtual Peripheral enabling technology in the SX processor is its deterministic nature. Every instruction in the SX is 100% deterministic, meaning all instructions execute in a predetermined number of clock cycles (1 to 3 clock cycles) and the interrupt latency is fixed (3 cycles for an internal interrupt, 5 cycles for an external interrupt.)

There is really only one true benefit to a deterministic architecture, but it is a big one: exact timing. Since all Virtual Peripherals run in the "background" of the main application software, they must run in an interrupt service routine. The deterministic nature of the SX gives the interrupt service routine an exact frequency of execution, and all Virtual Peripherals, whether a serial bus interface, a timer, or a DTMF generator, can be based on this exact, jitter-free frequency.

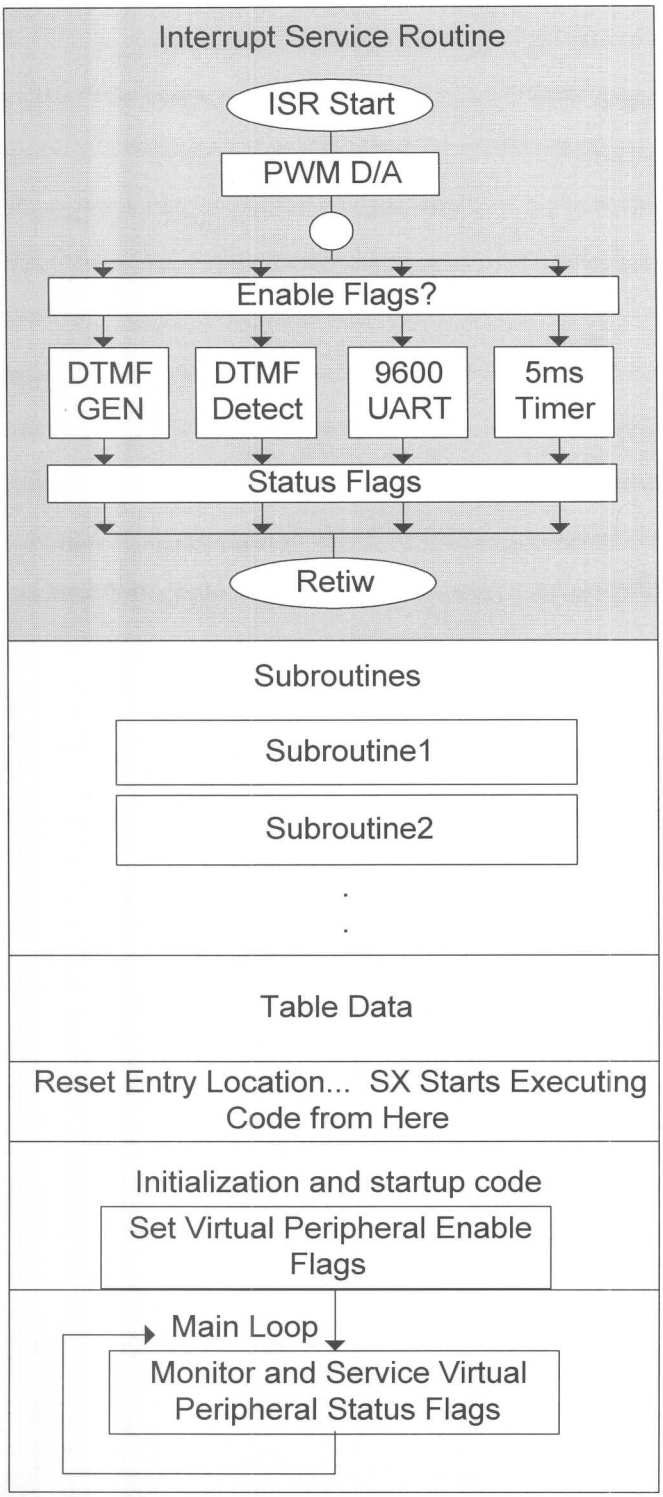
Running Virtual Peripherals that interface to the application software the same way a hardware peripheral does requires that they run with minimal intervention from the application software. The application software simply sets or clears flags, loads a few registers, and lets the Virtual Peripheral do the work. For instance, with an A/D Virtual Peripheral, there is no interaction needed from the application software while the conversion is taking place. When the A/D Virtual Peripheral has finished a conversion, it would set a flag in RAM. The application

software will be running it's main loop, and will get the newly converted value from RAM when it sees that the flag in RAM has been set.

The interrupt service routine is always set up to run at an exact timing interval. For most Virtual Peripherals, the timing can be arbitrary, as long as the sample rate is high enough to accomplish the desired task. Some Virtual Peripherals need to have fairly specific sample rates, such as with a UART Virtual Peripheral which has very specific timings… 1200bps, 2400bps, 9600bps, 19200bps, 38400bps, 57600bps, 115200bps, etc. To keep the interrupt service routine code that performs the UART sampling simple, the interrupt service routine must run at a timing that is a multiple of the standard UART speeds. Virtual Peripherals for the SX usually run from internal interrupts, triggered when the RTCC register (real-time-clock-counter) rolls over from 255 to 0. The timing for interrupts in the SX is set by the "RETIW" value. RETIW means "Return from interrupt with value in W added to the RTCC register." See section ??? for a detailed description of setting up the interrupt rates.

After the Virtual Peripherals have been properly set up, the remaining task is to set up the main loop of the program and its subroutines to properly interface with the Virtual Peripherals and perform the tasks of the program.

Structure of an Application with Virtual Peripherals



The Interrupt Service Routine is the key to Virtual Peripherals that appear transparent to the main program. It runs at a specific frequency and services all of the Virtual Peripherals. See page 21 for more information on setting up the Interrupt Service Routine.

To keep the format of all Virtual Peripherals consistent, the subroutines should follow the end of the interrupt service routine.

Table data like strings can be stored after the subroutines.

The entry point for the source code (Where the SX begins running the program on reset) should be right before the main loop at the end of the source code.

The main loop or the main program logic of the source code should be located at the end of the source code, where it is the easiest to find. The main program sets the enable flag for a Virtual Peripheral to start that Virtual Peripheral running in the Interrupt Service Routine. Virtual Peripherals will set flags in RAM to indicate to the Main Program that a specific function has been completed, such as the 5ms timer expiring.

RAM

Use the following standard label names for global variables in Virtual Peripherals and applications released by Scenix:

```
flags0          equ   global_org+0      ; semaphore register reserved
                                        ; for flag bits.
isrTemp0        equ   global_org+1      ; temporary register reserved
                                        ; for use by the interrupt-
                                        ; service routine, ISR VP's
localTemp0      equ   global_org+2      ; temporary register reserved
                                        ; for use by subroutines
localTemp1      equ   global_org+3      ; temporary register reserved
                                        ; for use by subroutines
localTemp2      equ   global_org+4      ; temporary register reserved
                                        ; for use by the main program
```

flags0 stores bit-wise operators like flags and function-enabling bits (semaphores).

isrTemp0 is for use ONLY by the interrupt service routine as a global register.

localTemp0 will be the temporary register used the most, so routines that are only nested once can destroy this register. It is never guaranteed to maintain its data from routine to routine

localTemp1 will be used by the second nested level, or when a routine needs more than one temporary global register.

localTemp2 follows along the same lines as localTemp1, but is used even less often by deeper nested routines or as a mainline loop counter, since the other Temp registers will probably be destroyed by the routines called by the mainline.

The documentation for each subroutine will specify which localTemp register it destroys, and which localTemp registers are destroyed by routines nested below this one.

Write Virtual Peripherals so they make use of no global RAM locations other than the definitions above.

If additional temporary registers are needed, they can be called localTemp3 or flags1, etc.

LABELS:

Keep all labels under two tabs in length.

Use Hungarian Notation for all labels. Example:

RS232_receive becomes *rs232Receive*

Prefix all RAM locations and constants for a specific Virtual Peripheral by a standard, truncated version of that Virtual Peripheral's name. Example:

rxByte ds 1 becomes rs232RxByte ds 1

Left justify all equates and defines, and group all of them into the "Equates and Definitions" area of the source code.

Standardized Directives

Create a set of standard directives for all VP's. A tradeoff is that some VP's may become less code-efficient or speed-efficient.

Standard Directives:

- OPTIONX enabled
- STACKX enabled
- CARRYX disabled - If a routine cannot run without CARRYX (like hard-core math), the fact that CARRYX is necessary must be well documented.
- TURBO mode

Standard !Option Setup

- WREG enabled (bit seven of !OPTION = 0) so W is accessible as a file register in location $01, rather than RTCC.
- If possible, a routine that needs to access the RTCC register in location $01 should set !option.7 before executing and, when finish, clear it again to access the WREG in location $01.

Define the default !option set-up as

```
OPTIONSETUP             equ     RTCC_PS_OFF|PS_111
```

Initialization code for WREG in location $01:

```
                mov     w,#OPTIONSETUP          ; setup option register for RTCC interrupts
                mov     !option,w               ; enabled and no prescaler.
                jmp     @main
```

Routines accessing the RTCC register in location $01:

```
                mov     !option, #(OPTIONSETUP | RTCC_ON); enable direct access of the RTCC
                                                          ; by setting !option.7
                mov     !rb,RTCC                ; This code accomplishes absolutely nothing,
                jmp     :yourmom                ; but it accesses the RTCC register
:yourmom        mov     RTCC,rc
                mov     !option, #OPTIONSETUP   ; go back to the option register's default.
```

After accessing the RTCC register, this routine sets the option register back to its default state. If an exception must be made for speed purposes, it should be well documented that the routine needs the option register set up to access the RTCC directly to run correctly.

Standard Mode Register Setup

In mainline (interruptible) program, never assume the value in the mode register, and always update it before using it.

In the Interrupt Service Routine, routines changing the mode register must change it back before exiting. The isrTemp register can be used to store and restore the previous state of the mode register. Example:

```
    mov   w,m           ; save mode register in isrTemp
    mov   isrTemp,w
    mov   w,#$1f        ; change mode register
    mov   m,w
    mov   !rb,#0        ; change port RB to all outputs
    mov   w,isrTemp     ; restore mode register
    mov   m,w
```

SX28/SX52 Compatibility

Use MACROs or IFDEF/IFNDEF statements to make portions of incompatible code switch in and out for SX28 and SX52.

Example of using an IFDEF statement for SX18/28 portability to SX52 and vice-versa:

```
        ;-------------------------------------------------------------------------------
        ;VP: 62-byte buffer
        ; Subroutine:  Store W in buffer[pushIndex++]
        ;       INPUTS:         data to store in W
        ;       OUTPUTS:        data stored in buffer[pushIndex++]
        ;       CHANGES:        localTemp1,pushIndex,buffer[pushIndex]
        ;-------------------------------------------------------------------------------
        bufferPush
                mov     localTemp1,w
                _bank   buffer
                mov     fsr,pushIndex
                mov     indf,localTemp1
                _bank   buffer
                inc     pushIndex
        IFDEF   SX28
                setb    pushIndex.4             ; keep even bank if SX28/18
        ENDIF
        IFDEF   SX18
                setb    pushIndex.4
        ENDIF
                retp
```

Since the RAM in the SX-52 is stored in contiguous banks, and in the SX-28 the banks are separated by $20, the IFDEF above will conditionally compile the setb instruction, allowing the pointer to memory to skip the non-existant banks in the SX-28.

Macros help keep the source code clean-looking… Example of a good MACRO for incrementing pointers to RAM:

```
        ;*****************************************************************
        ; INCP/DECP macros for incrementing pointers to RAM
        ;*****************************************************************
INCP    macro   1               ; Increments a pointer to RAM
        inc     \1
    IFNDEF  SX48_52
        setb    \1.4            ; If SX18 or SX28, keep bit 4 of the pointer = 1
    ENDIF                       ; to jump from $1f to $30, etc.
endm

DECP    macro   1               ; Decrements a pointer to RAM
    IFDEF   SX48_52
        dec     \1
    ELSE
        clrb    \1.4            ; If SX18 or SX28, forces rollover to next bank
        dec     \1              ; if it rolls over.  (Skips banks with bit 4 = 0)
        setb    \1.4            ; Eg:  $30 --> $20 --> $1f --> $1f
    ENDIF                       ; AND: $31 --> $21 --> $20 --> $30
endm
```

SX28/SX52 Compatibility

Using the INCP macro made the buffering source code easier to read…

```
    ;-------------------------------------------------------------------------------
   ;VP: 62-byte buffer
   ; Subroutine:  Store W in buffer[pushIndex++]
   ;   INPUTS:        data to store in W
   ;   OUTPUTS:       data stored in buffer[pushIndex++]
   ;   CHANGES:       localTemp1,pushIndex,buffer[pushIndex]
   ;-------------------------------------------------------------------------------
   bufferPush
       mov     localTemp1,w
       _bank   buffer
       mov     fsr,pushIndex
       mov     indf,localTemp1
       _bank   buffer
       INCP    pushIndex              ; Smart-Increment of the pointer to RAM
       retp
```

BANK 0… Location $10 to $1f

Since the SX48/52 can only access memory locations $10 to $1f directly, use these locations only as a last resort in programs written for the SX18/28, for compatibility with the SX48/52.

Extra RAM in the SX48/52

Since the SX18/28 only has half of the RAM of the SX48/52, avoid use of the extra RAM in programs written for the SX48/52 to remain compatible with the SX18/28.

Lookup Tables (Jump Tables)

Routines should be clearly marked as needing to be completely within the first half of the page. Lookup tables that may be called by the programmer's own program should have protection against the table extending into the second half of a page. This can be done with the assistance of macros. If every piece of code that includes tables uses a tableStart and tableEnd definition in the table, these macros will generate the error message "ERROR: Must be located in the first half of a page." when the table is misplaced.

```
       ;*******************************************************************************
       ; Error generating macros
       ;*******************************************************************************

tableStart     macro 0                ; Generates an error message if code that MUST be in
                                      ; the first half of a page is moved into the second
                                      ; half.
       if $ & $100
              ERROR  'Must be located in the first half of a page.'
       endif
       endm

tableEnd       macro 0                ; Generates an error message if code that MUST be in
                                      ; the first half of a page is moved into the second
                                      ; half.
       if $ & $100
              ERROR  'Must be located in the first half of a page.'
       endif
       endm
```

An assembler may implement this function in the future if these standard tableStart and tableEnd definitions are used.

This:

```
;********************
jmp_table_1
      add   pc,w
      jmp   routine_1
      jmp   routine_2
      jmp   routine_3
      jmp   routine_4
;********************
```

Becomes This:

```
;********************
jmp_table_1
; The code between the
; tableStart and tableEnd
; statements MUST be
; completely within the first
; half of a page.  The routines
; it is jumping to must be in
; the same page as this table.
tableStart
      add   pc,w
      jmp   routine_1
      jmp   routine_2
      jmp   routine_3
      jmp   routine_4
tableEnd
;********************
```

Note that the table must be in same page as the call to that table.

Indicate when a routine is program or data-memory-location dependent

Other routines that are program-memory dependent MUST also be clearly marked:

```
;*************************************************************************
; Subroutine - Send string pointed to by address in W register
;              Strings MUST be located completely within program memory space from $300
;              to $3ff
; INPUTS:
;       w       -       The address of a null-terminated string in program
;                       memory
; OUTPUTS:
;       outputs the string via. RS-232
;*************************************************************************
```

If possible, defines or equates can be used to simplify this process…

```
                mov     m,#3            ; move upper nibble of address of strings into m
```

becomes

```
                mov     m,#STRINGS_ORG>>8; move upper nibble of address of strings into m
```

Now, as long as the STRINGS_ORG label precedes the strings, this subroutine will work properly, regardless of where the strings are located.

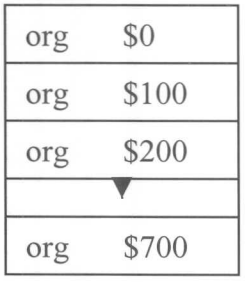
```
;*************************************************************************************
; String Data
;*************************************************************************************
org     STRINGS_ORG             ; This label defines where strings are kept in program
                                ; space.  All of the following strings must be within the
                                ; same 1/2 page of program memory for send_string to work,
                                ; and they must be preceded by this label.


_hello          dw      13,10,'V.23 Transmit (Originate Mode) 2.00',0
_FSK            dw      13,10,'Transmitting 75bps FSK >',0
```

For routines with very location-specific data memory definitions, there should be ample documentation to indicate that data memory cannot be moved around arbitrarily. Wherever possible, location-specific routines should be avoided.

Avoiding Page Boundaries

To ensure that several Virtual Peripherals, when pasted together, do not cross a page boundary without the integrator's knowledge, put an ORG statement with one instruction at every page boundary. This will generate an error if a pasted subroutine moves another subroutine to a page boundary.



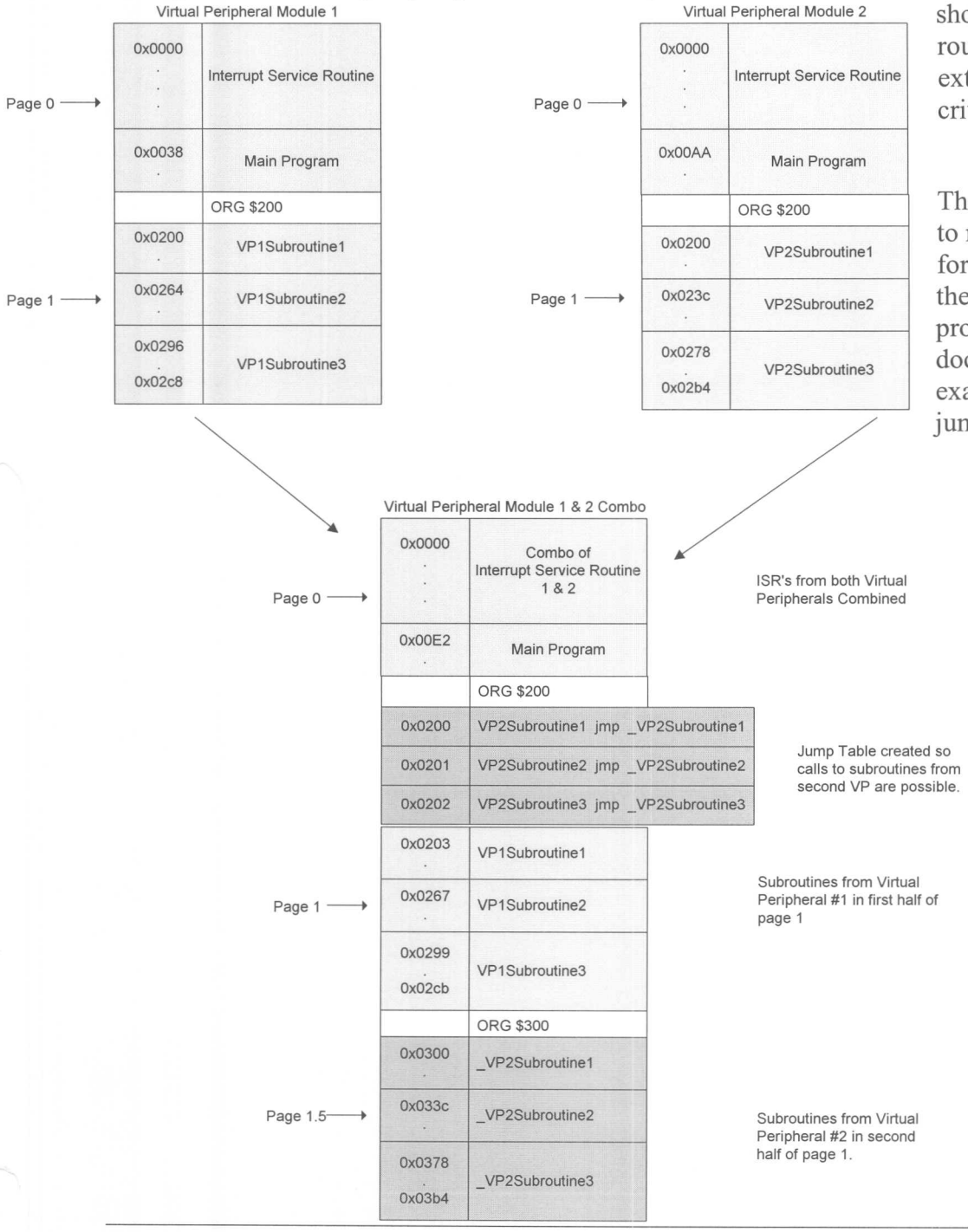

```
subroutines/program code...

org     $400            ; Even though there's no program,
        jmp     $       ; put code here to generate an error
                        ; if the code before it crosses a
                       ; page boundary.
org     $500
        jmp     $
etc...
```

Jump Tables: Making Subroutines Callable in the Second Page of Program Memory

If two Virtual Peripherals are integrated together, and the subroutines for each Virtual Peripheral are to be placed into the same page, the callable subroutines from one of the Virtual Peripherals may need to be moved to the second half of a page. The problem this poses is that labels in the second half of a page can only be jumped to and not called. The solution is to create a JUMP TABLE for the routines in the second half of the page. Unfortunately, the compromise is that calling a routine through a jump table adds a 3-cycle latency to the subroutine call, and therefore should only be used for routines that are not extremely speed-critical.

There is no simple way to make this job easier for the integrator, so the only solution is to provide ample documentation and examples on creating jump tables.



Define the ORG statements for each VP at the top of the code. ORG statements in the code use the pre-defined values, rather than literal values.

Place a table at the top of the source code with the starting addresses of all VP's. This will lend itself to separating the VP's into separate source files and creating a linker.

```
UART_SUBS_ORG          equ     $300
I2C_SUBS_ORG           equ     $400
I2C_ISR_ORG                  equ     $600
```

This would be an ideal place to indicate whether each segment is moveable or not.

Now, instead of using the literal values, use the defined values.

```
org     UART_SUBS
```

For smaller VP's, just use PAGE2_ORG, etc.

Example from included code:

```
;*****************************************************************************
; Program memory ORG defines
;*****************************************************************************
INTERRUPT_ORG          equ     $0      ; Interrupt must always start at location zero
INTERRUPT_ORG2         equ     $100    ; Some more of the ISR is stored in location $100
RESET_ENTRY_ORG        equ     $1FB    ; The program will jump here on reset.
SUBROUTINES_ORG        equ     $200    ; The subroutines are in this location
STRINGS_ORG            equ     $300    ; The strings are in location $300
PAGE3_ORG              equ     $400    ; Page 3 is empty
MAIN_PROGRAM_ORG       equ     $600    ; The main program is in the last page of program
                                       ; memory.
```

And…

```
;*****************************************************************************
       org     INTERRUPT_ORG                ; First location in program memory.
;-----------------------------------------------------------------------------
; Interrupt Service Routine
;-----------------------------------------------------------------------------
; Note: The interrupt code must always originate at address $0.
;
; Interrupt Frequency = (Cycle Frequency / -(retiw value))  For example:
; With a retiw value of -163 and an oscillator frequency of 50MHz, this
; code runs every 3.26us.
;-----------------------------------------------------------------------------
```

Use Small Amounts of SX Horsepower

Instead of running the Virtual Peripheral on every interrupt, try to write it so it can run on every 4th or 8th interrupt. This makes integrating the Virtual Peripheral with many other Virtual Peripherals much less likely to overflow the number of cycles available for each interrupt, because the Interrupt Service Routine need only run one thread at a time.

Document How to Calculate the Virtual Peripherals' Constants

Whenever possible, VP's should be written so that they are scaleable to work with virtually any interrupt rate. VP's that are written like this include the A/D and D/A converters, the timers, FSK and DTMF generation and detection, LCD interface, and many others. Only the resolution and/or timing constants change as the interrupt rate changes.

Some VP's, however, require very specific interrupt rates. One perfect example of this is the UART VP, which MUST be run at a 2^n multiple of the desired UART rate. In this case, it must be made very clear to the programmer how to calculate the interrupt rates for all VP's, so that VP's like the UART can be chosen as the determining factor, and all other VP's can have their constants re-calculated for the chosen rate.

```
;       19200 baud
;               baud_bit        =       4                       ;for 19200 baud
;               start_delay     =       16+8+1                  ; "     "       "
;               int_period      =       163                     ; "     "       "
```

This type of definition gives the programmer no idea what to do if he wants to change the interrupt rate, whereas the following makes the change much more obvious:

```
;       Execution rate/16
;               baud_bit        =       4                       ; For a baud rate of FS/16
;               start_delay     =       16+8+1                    ; "     "       "
;       Execution rate/8
;               baud_bit        =       3                       ; For a baud rate of FS/8
;               start_delay     =       8+4+1                    ; "     "       "
;       Execution rate/4
;               baud_bit        =       2                       ; For a baud rate of FS/4
;               start_delay     =       4+2+1                    ; "     "       "
```

This clears up what the defines do to the UART speeds, and that the UART speed is tightly tied in to the sampling rate (read: interrupt rate) Plus, if the programmer wants a slower UART, or a slower interrupt rate, or both, the change more intuitive.

If possible, try to let the compiler calculate the constant for the programmer, as in this example:

```
Fs = 9600                               ; sampling frequency for DTMF detection
Bits = 65536                            ; 2^16 is the value of the phase accumulator

f697_l          equ     ((Bits * 697)/Fs) & $0ff
f697_h          equ     ((Bits * 697)/Fs) >> 8
```

With this example, the constant used to generate a 697Hz signal is generated on the fly, as Fs changes. If the programmer wants to speed the execution rate of the frequency generating VP, he just changes it's execution rate in the ISR, and scales the FS constant accordingly.

As long as the execution rates of each of the VP's are multiples of one another, they can all use the same retiw value.

Unfortunately, the downfall of having only one jitter-free interrupt is that running Virtual Peripherals at rates that don't have a common denominator such as 115kHz and 250kHz is not possible.

Make V.P.'s look and act as modules:

- Keep each module SHORT. The entire routine, beginning to end, should fit in the editor's window without scrolling.
- Indicate whether each ISR routine is variable-length or fixed-length in header
- Indicate the worst-case cycle time for each ISR routine in header.
  - Use only Scenix mnemonics as defined in the datasheet in final, releasable source code.
- Only ONE way in and out of each routine.
- Callable ISR Routines, with bank instruction for routine performed before the call..
- Maximum ISR Call-Nesting: two levels, leaving nest-able levels for the mainline program.
- As a rule, always return with a retp instruction rather than a ret instruction.

Each VP should have all of its pieces labeled: RAM, Constants, ISR, Subroutines.

- Each piece of the Virtual Peripheral should be LABELED CLEARLY to show which Virtual Peripheral it belongs to. This includes RAM allocations, Program Constants, Virtual Peripheral (ISR) routines, and call-able subroutines.
- The labels should be machine readable, to enable a smart software tool in the future that can automatically cut and paste VP's.
  - Start the VP module with
    ;VP_begin <moduleVP>
    And end it with
    ;VP_end ; End cut/paste ....

```
;VP_begin RS232Receive

RS232RxBank     =       $

RS232RxCount    ds      1                       ;number of bits received
RS232RxDivide   ds      1                       ;receive timing counter
RS232RxByte     ds      1                       ;buffer for incoming byte

;VP_end                                 ; End cut/paste for this Virtual Peripheral
```

Place the Main Program at the End of the Source Code

The main program operation should be easy to find, so place it at the end of the program code. This means that if the first page is used for anything other than main program source code, a reset_entry must be placed in the first page, along with a 'page' instruction and a 'jump' instruction to the beginning of the main program.

Example:

```
;*******************************************************************************************
org     RESET_ENTRY_ORG
;*******************************************************************************************
;-------------------------------------------------------------------------------
reset_entry                                     ; Program starts here on power-up
        page    _reset_entry
        jmp     _reset_entry
;-------------------------------------------------------------------------------
```

Then… at the start of the main routine on another page…

```
;*******************************************************************************************
org     MAIN_PROGRAM_ORG
;*******************************************************************************************
;*******************************************************************************************
; RESET VECTOR
;*******************************************************************************************
        ;***************************************************************************
        ; Program execution begins here on power-up or after a reset
        ;***************************************************************************
_reset_entry
        ; program start up source code here
```

Port Access

To ease integrating multiple Virtual Peripherals that all need access to the same ports,

- Use pin definitions rather than port definitions.
- all port accesses should be made through symbolic names. Example:

setb rb.6 becomes setb LEDPin

Port Direction Registers/Mode-Addressable Registers

When a Virtual Peripheral must dynamically change a port direction register, it should do this through the use of a **port direction buffer**. The port direction register stores the initialized state of the port direction register, and any changes made to the port direction register are made to the buffer first, and the buffer is then written to the port direction register.

If two Virtual Peripherals are combined, and both need to dynamically modify the same port direction register, they instead operate on the buffer for that port's direction register and the buffer will then be written to the port direction register. Use banked RAM and standardized names for port direction register buffers:

```
portBufBank equ   $
RADirBuf    ds    1
RBDirBuf    ds    1
RCDirBuf    ds    1
RDDirBuf    ds    1
REDirBuf    ds    1
```

These rules apply to other special mode-register addressable registers, such as the pull-up enable registers, etc.

The interrupt service routine should always be defined in location $0 to $1ff. Extra pages used only if necessary.

Keeps the ISR structure compact and easy to read.

Ensure the worst case cycle count for any ISR thread does not exceed the number of cycles between interrupts.

Exceeding the interrupt timing will cause the processor to miss an interrupt, throwing off the timing of every interrupt-driven Virtual Peripheral in the program.

Designing the Interrupt Routine

Making your SX interrupt at your desired interval with the desired oscillator may be one of the most confusing parts of designing an interrupt-driven Virtual Peripheral, so here it is explained in a few steps:

- Choose a desired interrupt frequency (irqFreq) based on the Virtual Peripherals you want to run.
  - Example: Choose 230.4kHz for 4 times over-sampling on a 57.6kbps UART

- Choose an oscillator frequency (oscFreq.) Higher sample-rate Virtual Peripherals will require higher oscillator frequencies. If power is not an issue for your design, a 50MHz-oscillator frequency is a safe bet for almost all Virtual Peripherals.
  - Example: Choose 50MHz

- Divide your oscillator frequency by your interrupt frequency. This is your ideal RETIW value.
  - Calculate RETIW value = (oscFreq/irqFreq)
  - Calculate 50MHz/230.4kHz = 217.01

- Round your RETIW value to the nearest integer value between 0 and 255.
  - Round RETIW value to an integer
  - Round to 217
  - If the number exceeds 255, then slow down the RTCC by enabling its prescaler (reducing its time between RTCC increments by factors of 2), or choose a lower oscillator frequency. If the number is 90 or less, there may not be enough time to service each interrupt, so increase the oscillator frequency or decrease the interrupt frequency.

- Calculate your actual interrupt frequency to see if it is close enough to your desired interrupt frequency by dividing the oscillator frequency by the RETIW value
  - Actual Frequency = (oscFreq/RETIWVal * prescaler)
  - Check Actual Frequency = 50,000,000Hz/217 = 230.415kHz ≅ 230.4kHz
  - If the difference between the desired interrupt frequency and the actual interrupt frequency is too much, try re-calculating with different oscillator frequencies.

Utilize the Multi-Threaded ISR Template.

- This method produces a FAR smaller worst-case cycle time count, and enables a larger number of VP's to run simultaneously. Also produces "empty" slots that future VP's can be copied and pasted into easily.
  - Create a simple multi-threaded Interrupt-Service-Routine template like the one on the next page.
  - Determine how often your tasks need to run. (9600bps UART can run well at a sampling rate of only 38400Hz, so don't run it faster than this.)
  - Strategically place each "module" into the threads of the ISR. If a module needs to be run more often, just call it's module at double the rate or quadruple the rate, etc....
  - Split complicated Virtual Peripherals into several modules, keeping the high-speed portions of the Virtual Peripherals as small and quick as possible, and run the more complicated, slower processing part of the Virtual Peripheral at a lower rate.
    - For example, in the Caller-ID detection program, the zero-cross-timer component of the Virtual Peripheral runs at double the speed of all of the other Virtual Peripherals, since it needs high resolution timing on the transitions on a pin. The other components of the Caller-ID Virtual Peripheral run at a slower rate, yet take a longer time to run when they are run. It is not necessary for them to run any faster, however, and doing so would increase the number of MIPS used by the FSK detection Virtual Peripheral with no added benefit. (See the block diagram of the Caller-ID detection Interrupt Service Routine on the following pages...)

Since the Interrupt Service Routine can now be viewed as a structure, made up of modules, it is easier for the user to increase and decrease the sampling rate by moving the modules around in the source code. Since only a few tasks are called in each interrupt, the flow of each interrupt is smaller and better understood. It is much simpler than an interrupt structure made up of many modules running consecutively, each jumping to the next.

Index of Following Pages:

Page 23: Block diagram of a complex application, simplified through the use of a multi-threaded ISR (14 Simultaneous VP's)
Page 24: Block diagram of an actual application following this document's specifications. (FSK Generation with 10 Simultaneous VP's)
Page 25: Block diagram of the Caller-ID detection application's Interrupt Service Routine.